BITS Pilani

ML SYSTEM OPTIMIZATION

Session 3

M1: Foundations

Modern Hardware Architectures

innovate achieve lead

BITS Pilani

1. *Multi-core CPUs*
2. *GPGPUs*
3. *GPU/CPU Clusters*
4. *Resource Management*

Multi-core CPUs

A **multi-core CPU** integrates multiple independent processing cores on a single chip.

Each Core Contains:

- Private **registers**
- **ALU (Arithmetic Logic Unit)**
- Often private **L1 cache**
- Shared **L2/L3 cache**
- Shared **main memory (RAM)**

Parallel Model → Shared Memory Model

All cores access the same global address space:

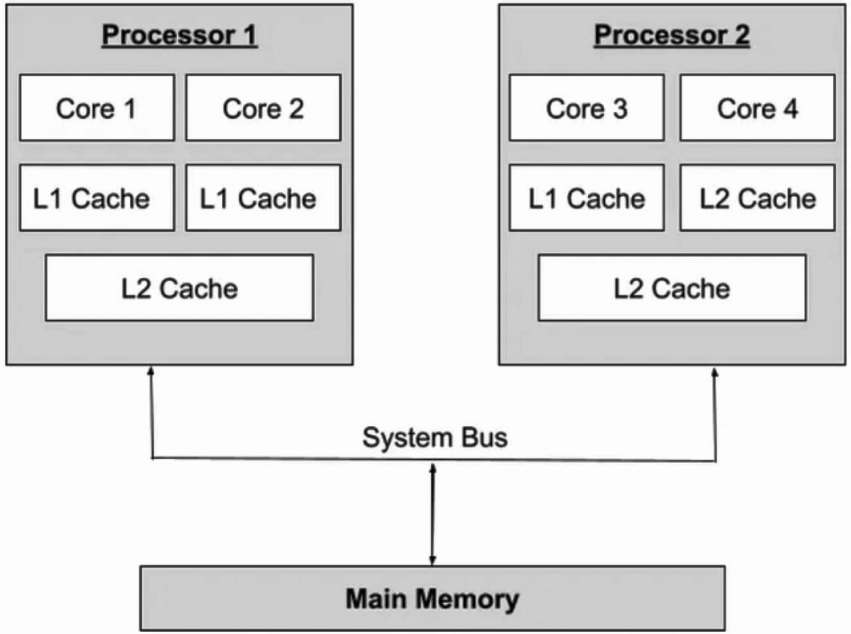
$$M_{global} \rightarrow \{Core_1, Core_2, \ldots, Core_N\}$$

This enables:

- Thread-based programming (e.g., OpenMP, Pthreads)
- Fast inter-thread communication (via shared variables)

Work Integrated Learning Programmes

Multi-core CPUs

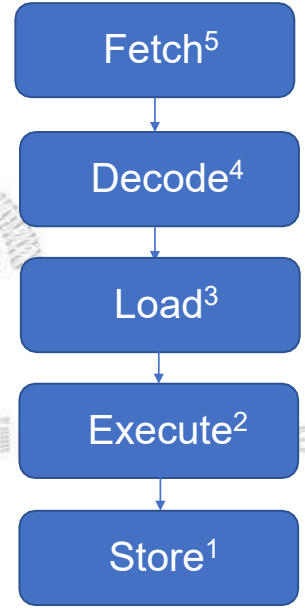


Multi-core CPUs - Instruction Pipeline

Given a sequence of instructions of the form:

- I1
- I2
- I3
- I4
- I5
- …

execution in a pipeline would appear thus ==>



If each stage takes 1 clock cycle, then throughput has increased:

- from 1 instruction per 5 cycles (sequential)
- to 1 instruction per 1 cycle (pipelined)

Q: What about Turn-around-Time aka response time?

Executing an Instruction

- Different stages of Instruction Execution:

Fetch Instruction	*Move the next instruction (tracked by PC) to a register*
Decode Instruction	*Identify Operator and (data) addresses*
Load (data)	*Move data into register (if needed)*
Execute	*Perform the operation*
Store (result)	*Move the resulting data to memory*

If separate circuitry is designed for each stage-
so that the stage take the same amount of time -
then a sequence of instructions can be executed in a pipeline

SIMD – Single Instruction, Multiple Data

Executes the **same instruction** across multiple data elements

Key Features:

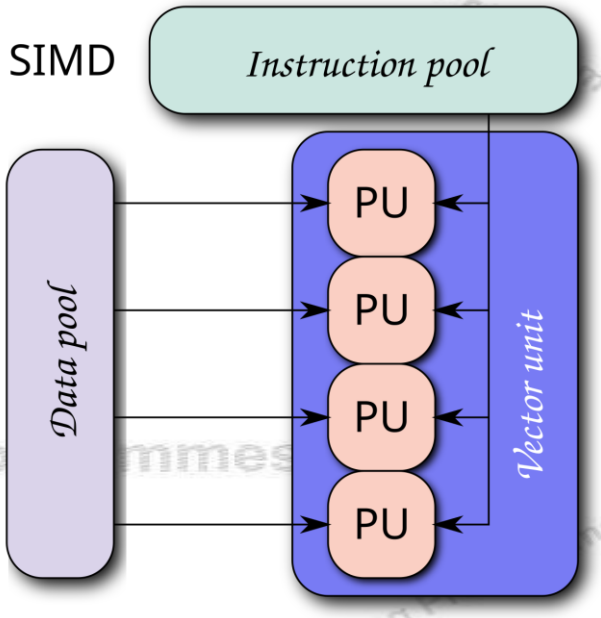
- Suitable for **data-parallel** applications
- Lockstep execution – all processing elements perform same operation simultaneously

Examples of Use:

- Vector and matrix operations
- Image filters, audio processing
- Scientific computation

Hardware Support:

- Intel AVX, ARM NEON, NVIDIA CUDA (warps/wavefronts)



MIMD – Multiple Instruction, Multiple Data

Each processor or core can execute **different instructions** on **different data**

Key Features:

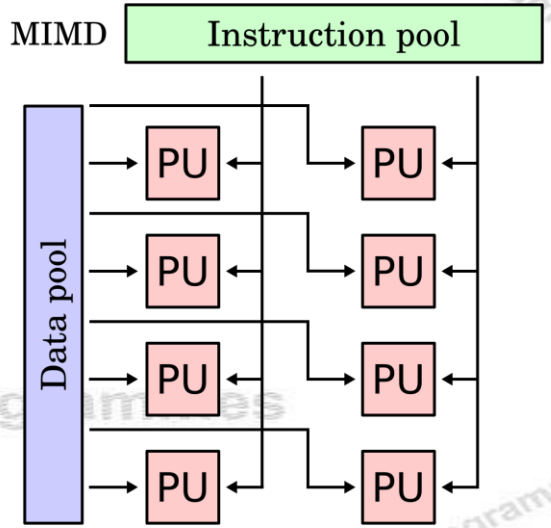
- Flexible and versatile execution model
- Ideal for task-parallel workloads

Examples of Use:

- Operating systems
- Web servers
- Multithreaded applications

Hardware Support:

- Multicore CPUs (Intel, AMD)
- Some heterogeneous compute platforms (e.g., ARM big.LITTLE)



SIMD vs. MIMD – Key Differences

Aspect	SIMD	MIMD
Instruction Stream	Single	Multiple
Data Stream	Multiple	Multiple
Synchronization	High (lockstep)	Low (independent threads)
Control Flow	Less flexible	Highly flexible
Hardware Examples	GPUs, Vector Units	Multicore CPUs, Clusters
Use Cases	Matrix ops, signal processing	Web servers, OS-level tasks
Efficiency	High for regular data structures	High for diverse workloads

Work Integrated Learning Programmes

Performance Model

If a program has:

- Parallel fraction = P
- Number of cores = N

Then theoretical speedup:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

- Barriers(1−P) → Sequential part (cannot be parallelized)
- P / N → Parallel part divided across N cores

$$T_{parallel} = T_{compute} + T_{synchronization} + T_{memory\ contention}$$

$T_{compute}$: Actual useful computation time.

$T_{synchronization}$: Time spent:

- Waiting for other threads
- Barriers
- Locks / coordination

T_{memory} : Time lost due to:

- Shared memory access conflicts
- Cache coherence traffic
- Limited memory bandwidth

Memory Hierarchy Impact

CPU performance depends heavily on **cache behavior**.

Access times:

$$T_{L1} < T_{L2} < T_{L3} < T_{RAM}$$

Since RAM access is much slower, cache performance dominates CPU speed

Effective memory access time (EMAT):

$$EMAT = H \cdot T_{cache} + (1 - H) \cdot T_{memory}$$

Where:

- H = cache hit rate
- T_{cache} = cache access time
- T_{memory} = RAM access time

Higher H → lower average memory delay → better CPU performance.

Work Integrated Learning Programmes

Multi-core CPUs

Multi-core CPUs are effective for:

- Task-level parallelism
- Moderate data parallelism
- Branch-heavy computations
- General-purpose workloads

Multi-core performance is determined by:

$$Performance = Parallel\ Speedup - Synchronization + Memory\ Overhead$$

Even with many cores, scalability depends on:

- Parallel fraction P
- Synchronization cost
- Memory behavior
- Cache efficiency

GPGPUs

A GPGPU is designed for **massive parallel throughput computing**.

It contains:

- Thousands of lightweight cores
- Organized into **Streaming Multiprocessors (SMs)**
- Each SM executes many threads concurrently

CPU	GPU
Few powerful cores	Thousands of simpler cores
Optimized for low latency	Optimized for high throughput
Complex control logic	Minimal control logic
Large cache	Smaller cache, higher bandwidth

Execution Model – SIMT

SIMT - Single Instruction, Multiple Threads

One instruction is executed by many threads simultaneously.
Example:

$$C[i]= A[i] + B[i] \; C[i]$$

If:

- n elements
- T threads

Each thread computes one index independently. Since operations are independent, they execute in parallel.
This model is ideal when:

- Same operation
- Different data elements

Memory Hierarchy in Multicore & GPGPU

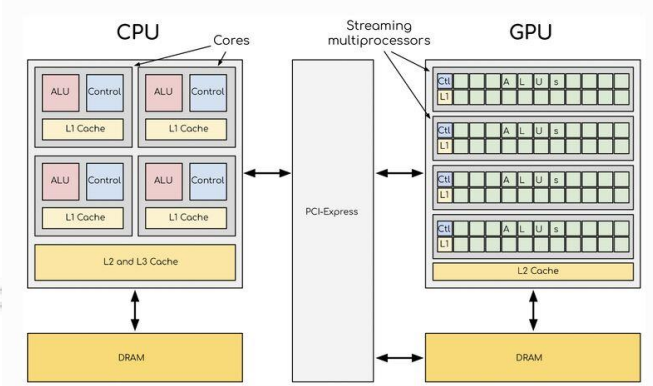
Purpose: Bridge speed gap between fast processors and slow memory

Hierarchy Levels:

- **Registers**: Fastest, smallest, processor-specific
- **L1, L2, L3 Cache**: On-chip, close to core, progressively larger and slower
- **Main Memory (RAM)**: Shared DRAM, slower than cache
- **Disk/SSD**: Persistent storage, very high latency

GPU-Specific:

- **Shared Memory**: Fast, limited, user-controlled
- **Global Memory**: Accessible by all threads, high latency
- **Constant/Texture Memory**: Specialized, cached memory for specific use cases



GPU Memory Hierarchy

GPU performance strongly depends on memory behavior.

Memory Types:

- **Registers (Fastest):** Private to each thread.
- **Shared Memory (Fast):**
  - Shared among threads within a block.
  - Low latency, programmer-controlled.
- **Global Memory (Slow):**
  - Accessible by all threads.
  - High latency (hundreds of cycles).

innovate achieve lead
Work Integrated Learning Programmes

GPU Memory Hierarchy

Performance Depends On:

Memory Coalescing

Adjacent threads should access adjacent memory locations:

Coalesced Access ⇒ Fewer memory transactions

Minimizing Global Memory Access

Use shared memory to reduce global access.

Maximizing Occupancy

Occupancy = Active threads / Maximum supported threads

Higher occupancy hides memory latency.

Work Integrated Learning Programmes

GPU Memory Hierarchy

GPU performance is dominated by:

Performance = Massive Thread Parallelism + High Memory Bandwidth

Unlike CPUs (latency-optimized), GPUs are:

Throughput-Optimized Processors

They perform best when:

- Large data size n
- High parallel fraction P
- Minimal branching
- Memory access is well-structured

Memory Bandwidth Considerations

Memory Bandwidth: Maximum rate at which data can be transferred between memory and processors

Why Important?

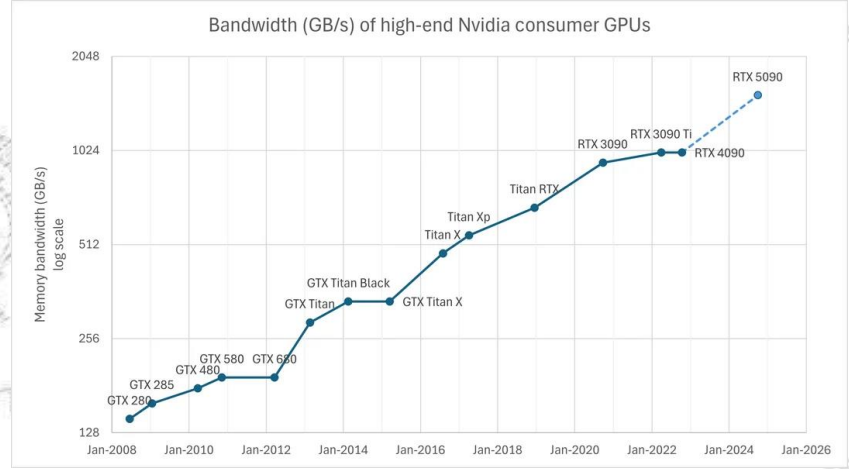
High computation speeds are meaningless without fast data access

Multicore CPUs:

- Shared bandwidth among cores → contention possible
- Cache reduces memory access latency

GPGPUs:

- Very high theoretical memory bandwidth
- Sensitive to **access patterns**
- **Coalesced access** (threads accessing adjacent memory) is critical for performance



Optimizing Memory Performance

For Multicore CPUs:

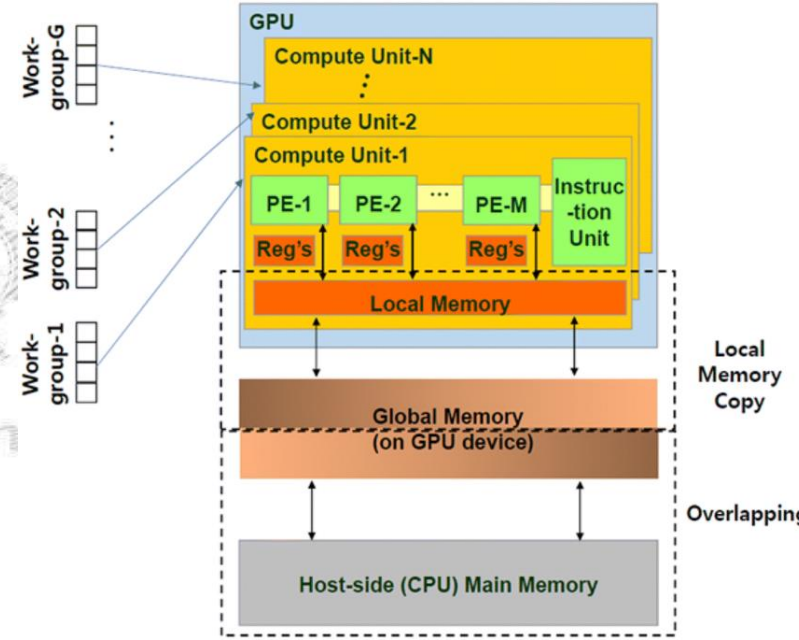
- Use **locality of reference** to benefit from caches
- Avoid **false sharing** – separate data written by different threads
- Use cache-aware data structures

For GPGPU:

- Align memory access for coalescing
- Utilize **shared memory** to avoid global memory latency
- Minimize **bank conflicts** in shared memory

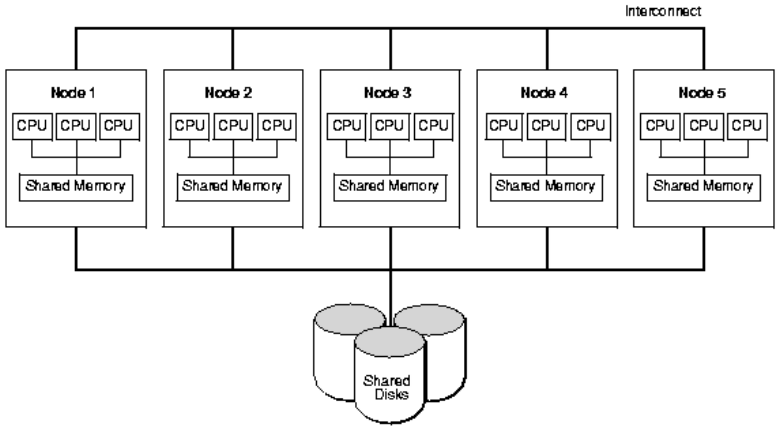
General Strategies:

- Balance **compute vs memory** loads
- Use **pipelining and tiling** techniques



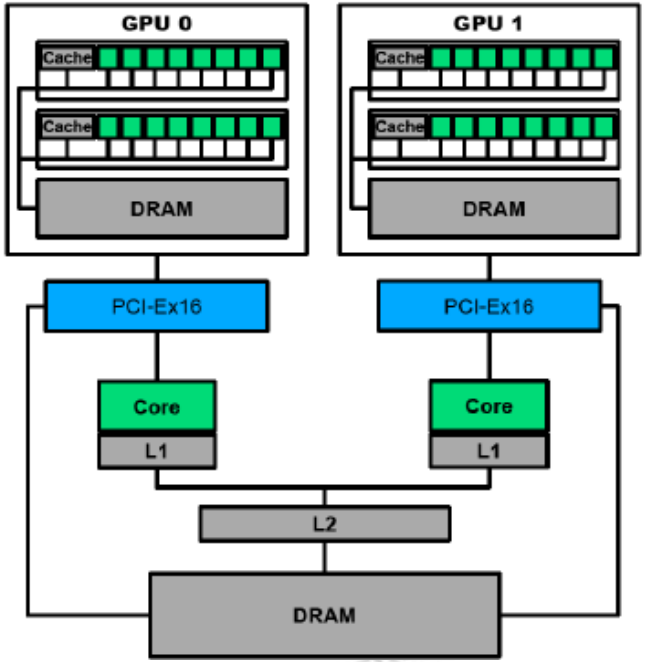
CPU-only Clusters

- Comprised solely of **multicore CPU-based nodes**.
- Suitable for:
  - Applications with control-heavy logic
  - Large in-memory databases
  - Distributed file systems and ETL pipelines
- **Benefits:**
  - More memory per node (better for large datasets)
  - Well-supported by traditional distributed frameworks (e.g., Hadoop, Spark, MPI)
  - Easier to manage and deploy
- **Limitations:**
  - Lower computational throughput
  - Slower performance in compute-intensive tasks like deep learning



GPU-Accelerated Clusters

- Consist of nodes with both CPUs and **one or more GPUs** (NVIDIA, AMD).
- Ideal for:
  - Training large-scale deep learning models (CNNs, Transformers)
  - Image/video processing, simulations
- **Benefits:**
  - Massive parallelism using thousands of CUDA cores
  - Faster matrix operations (e.g., GEMM, convolutions)
  - High floating-point throughput (TFLOPs range)
- **Limitations:**
  - Lower memory per GPU
  - Requires GPU-compatible libraries (e.g., TensorFlow GPU, CUDA, NCCL)
  - Higher hardware and maintenance cost



CPU-only vs GPU-Accelerated Clusters

Feature	CPU-only Clusters	GPU-Accelerated Clusters
Compute Units	Multicore CPUs	CPUs + Dedicated GPUs
Memory per Node	High	Lower (per GPU)
Programming Model	Threading, MPI	CUDA, OpenCL, TensorFlow, PyTorch
Power Consumption	Lower	Higher
Ideal Use Cases	Control-heavy tasks, I/O-bound jobs	Deep learning, simulations, HPC
Cost	Lower	Higher (GPUs, cooling systems, etc.)

Work Integrated Learning Programmes